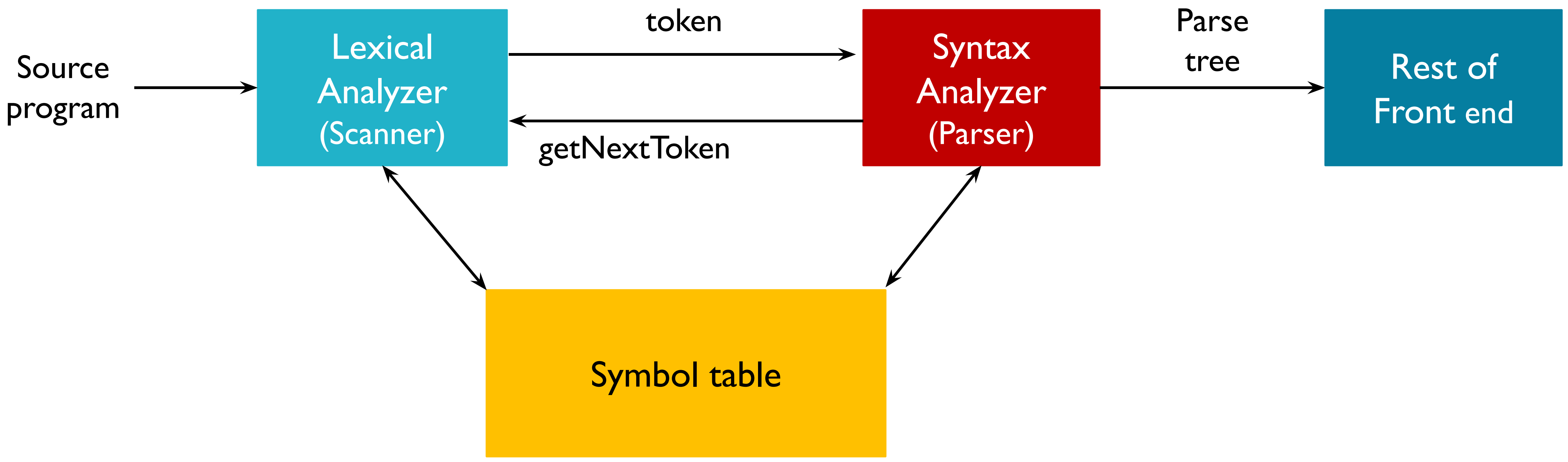
Syntax Analysis

Role of Parser



Errors

- Lexical – misspelling an identifier, keyword or operator
- Syntactic – arithmetic expression with unbalanced parenthesis
- Semantic – an operator applied to an incompatible operand
- Logical – an infinitely recursive call

- ***often much of the error detection and recovery in a compiler is centered around the syntax analysis phase

CFG (Context Free Grammar)

- CFG consists of terminals, nonterminals, start symbols and productions

- ❖ Terminals
  - Basic symbols from which strings are formed
  - “token name” is synonym for “terminal”

- ❖ Nonterminals
  - Syntactic variable that denote sets of strings

CFG (Context Free Grammar)

- CFG consists of terminals, nonterminals, start symbols and productions

- ❖ Start symbol
  - One nonterminal different from other
  - Set of strings it denotes is the language generated by the grammar
  - Its productions are listed first

- ❖ Production
  - Specify the manner in which the terminal and nonterminal can combine to form strings

CFG (Context Free Grammar)

- Production consist of
  - Nonterminal called the “head” or “left side”
  - Symbol →
  - “body” or “right side” consisting of zero or more terminals and nonterminals

CFG (Context Free Grammar)

- Grammar for arithmetic expression

expression	→	*expression + term*
expression	→	*expression - term*
expression	→	*term*
term	→	*term * factor*
term	→	*term / factor*
term	→	*factor*
factor	→	*(expression)*
factor	→	**id**

CFG (Context Free Grammar)

- Grammar for arithmetic expression

expr → *expr* *op* *expr*

expr → (*expr*)

expr → - expr

expr → id

op → +

op → -

op → *

op → /

Notational convention for grammar

- Symbols for terminals

Lowercase letters	a , b , c , … , z
Operator symbols	+ * / etc.
Punctuation symbols	, ; etc.
Digits	0 , 1 , 2 , … , 9
Boldface strings	**id** , **if** etc.

- Symbols for nonterminals

Uppercase letters	A , B , C , … , Z
S	usually indicated start symbol
Lowercase, italic names	*expr* , *stmt* etc.

Notational convention for grammar

- X , Y , Z represents grammar symbols
  either nonterminal or terminal

- u , v , … , z represents strings of terminals

- α , β , γ , represents strings of grammar symbols (terminal and/or nonterminal)

- $A\square\alpha_1$, $A\square\alpha_2$, … , $A\square\alpha_k$ may be written as
  $A\square\ \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_k$

- Unless stated, head of first production is start symbol

Language generated by grammar

- G : Grammar
  L(G) : Language generated by grammar G

- A language generated by CFG is called CFL (Context Free Language)

- Two grammar generate the same language, the grammars are said to be equivalent

Derivation

- Beginning with the start symbol, each rewriting step replaces a nonterminal by the body of one of its production

Grammar: E $\rightarrow$ E + E | id

String: id + id +id

Derivation: $\underline{E}$

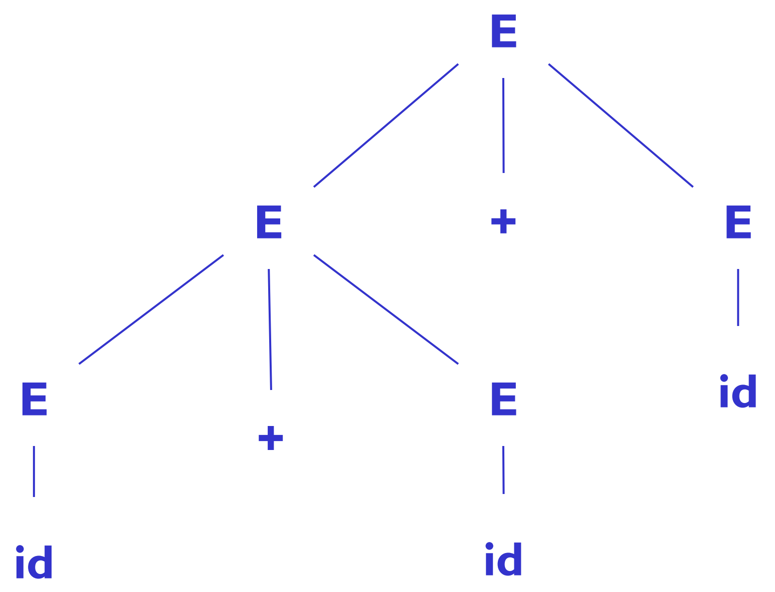
$\underline{E}$ + E

$\underline{E}$ + E + E

id + $\underline{E}$ + E

id + id + $\underline{E}$

id + id + id



Derivation

- $\Rightarrow$: derive in one step
- $\overset{*}{\Rightarrow}$: derive in zero or more step

Grammar (G) : E E + E | E * E | - E | (E) | id

Derive string : – (id)

$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(id)$

Can also be written as

$E \overset{*}{\Rightarrow} -(id)$

Derivation

- Lest most derivation
  - ❖ Left most nonterminal will be first replace by its production
- Right most derivation (canonical derivation)
  - ❖ Right most nonterminal will be first replace by its production

Grammar : $E \rightarrow E + E \mid E * E \mid -E \mid (E) \mid id$

String : $-(id + id)$

Left most derivation

$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(id+E) \Rightarrow -(id+id)$

Right most derivation

$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(E+id) \Rightarrow -(id+id)$

Reduction

- Specific substring matching with the production of nonterminal will be replaced by that nonterminal

Grammar: E → E + E | id

String: id + id +id

Derivation: $\underline{\text{id}}$ + id + id

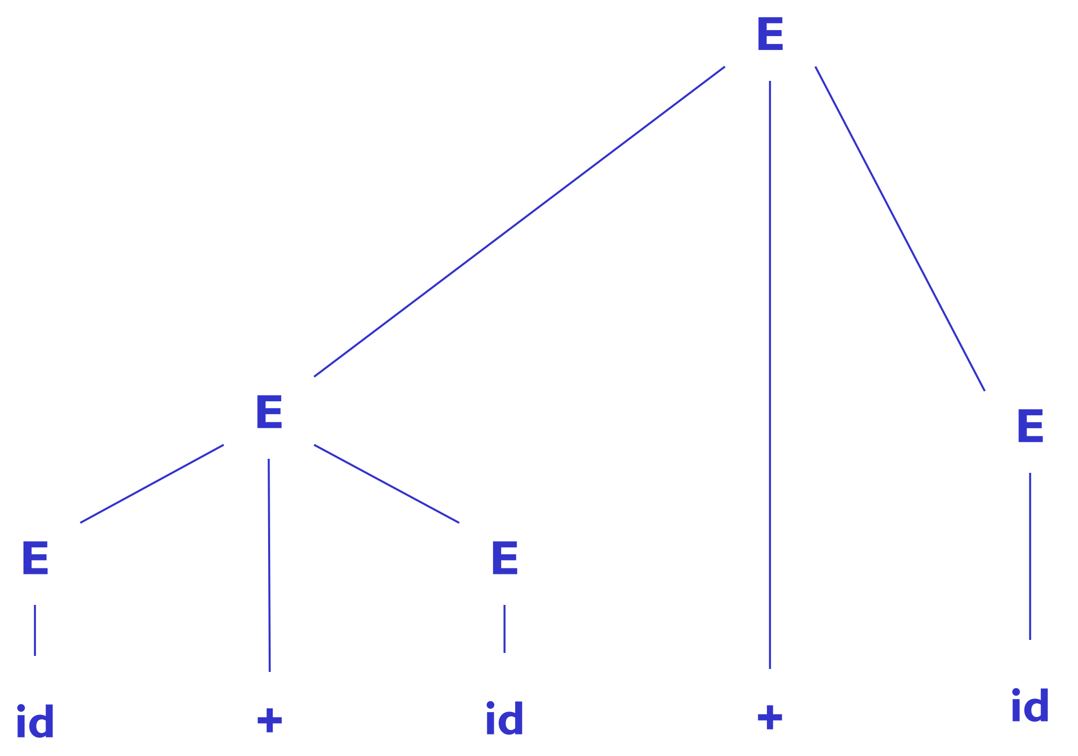
$\quad$ E + $\underline{\text{id}}$ + id

$\quad$ $\underline{\text{E + E}}$ + id

$\quad$ E + $\underline{\text{id}}$

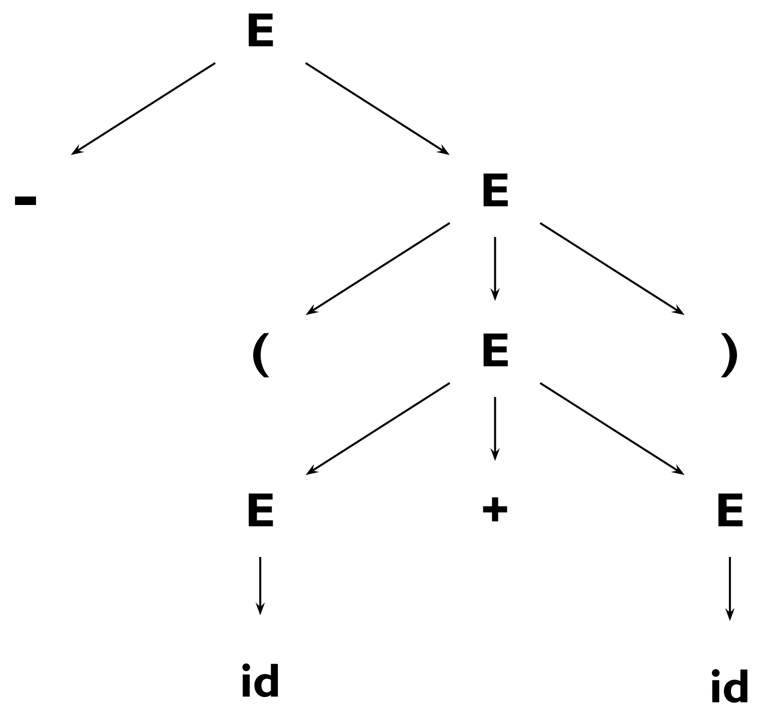
$\quad$ $\underline{\text{E + E}}$

$\quad$ E



Parse tree

- Graphical representation of derivation
- Parse tree for the string - (id + id) is

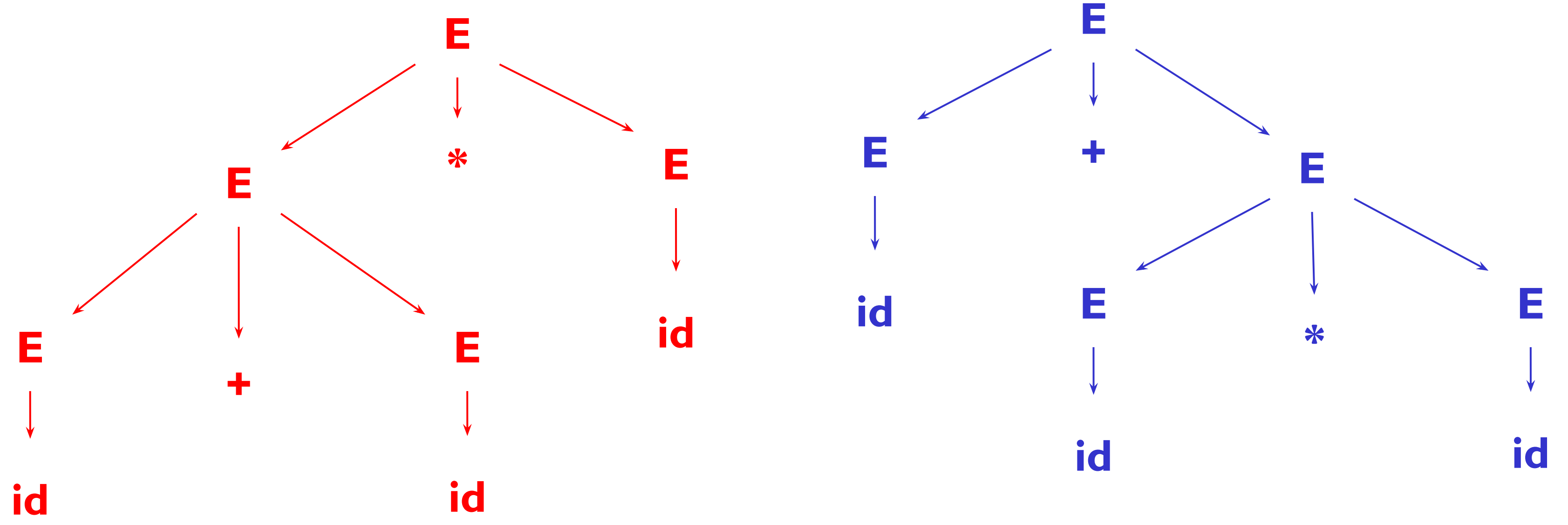


Ambiguity

- A grammar that produce more than one parse tree for some string is said to be ambiguous grammar
- more than one left most derivation or more than one right most derivation

Grammar:
E E + E | E * E | **id**

String :
id + id * id



CFG vs. RE

- Grammar are more powerful than RE
- Everything that can described by a RE can be described by a Grammar, but not vice-versa
- Every regular language is context free language but not vice-versa

CFG vs. RE

- RE : (a | b) * a b b

- Grammar :

  S $\rightarrow$ aX | aS | bS

  X $\rightarrow$ bY

  Y $\rightarrow$ bZ

  Z $\rightarrow$ ϵ

Left recursion

- A grammar is left recursive if it has a nonterminal A such that there is a derivation $A \overset{+}{\Rightarrow} A\alpha$ for some string α.

- Top-down parsing methods cannot handle left-recursive grammar, so, a transformation that eliminates left recursion is needed.

- Eliminate left recursion

i/p : Grammar with left recursion : $A \rightarrow A\ a \mid b$

o/p : Grammar without left recursion : $A \rightarrow b\ A'$
$A' \rightarrow a\ A' \mid \epsilon$

Left factoring

- Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing.

- If $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ are two productions of A and
  the input string begins with a nonempty string derived from α,
  we do not know whether to expand A to $\alpha\beta1$ or $\alpha\beta2$.

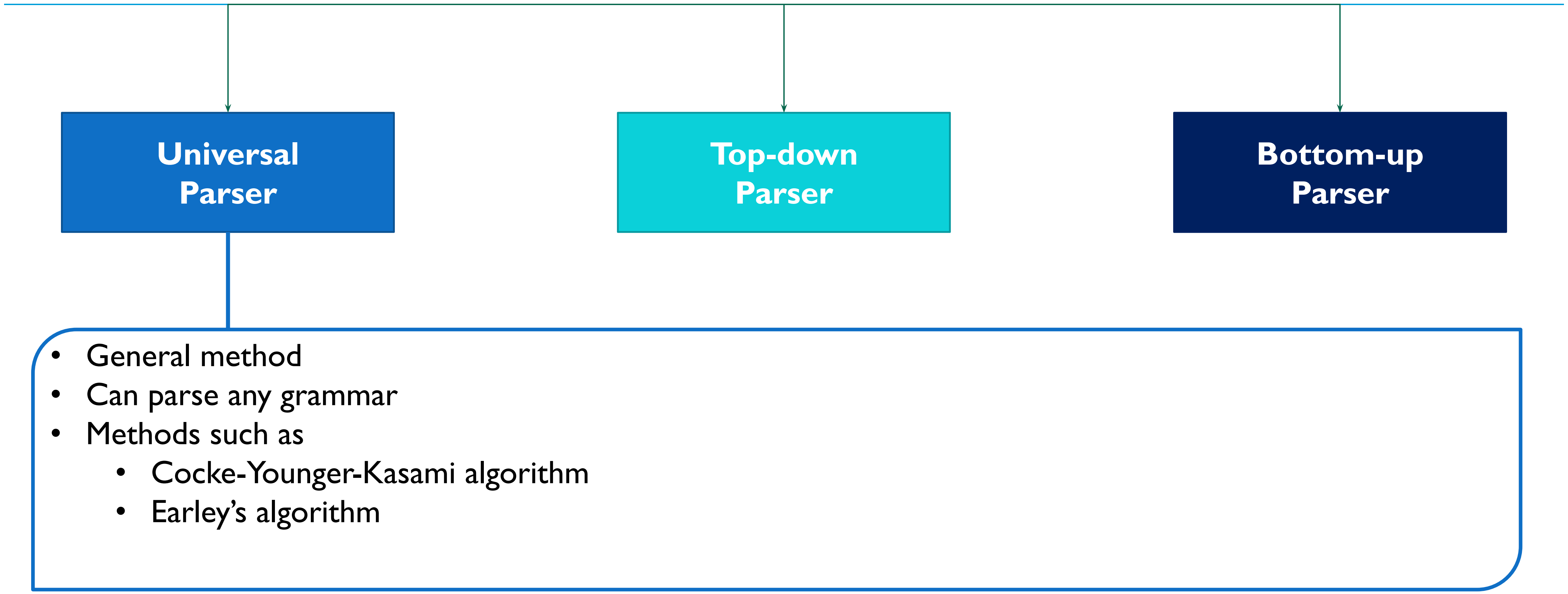
- Left factoring

i/p : Non left factored grammar : $A \rightarrow \alpha\ \beta_1 \mid \alpha\ \beta_2$

o/p : Left factored grammar : $A \rightarrow \alpha\ A'$

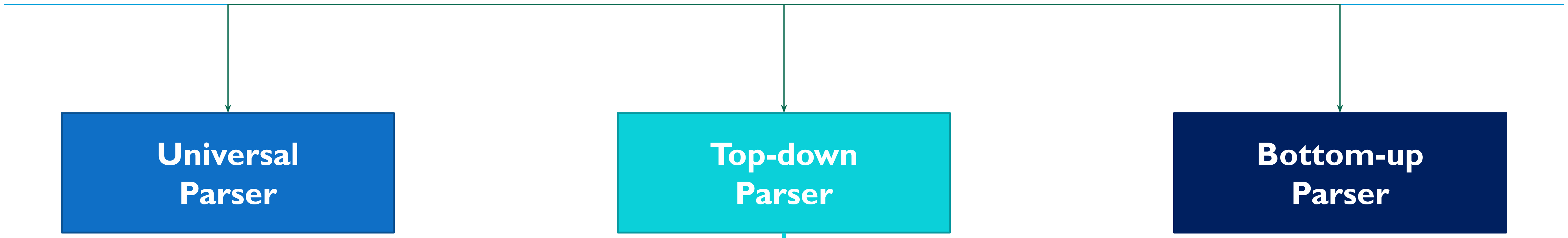
$A' \rightarrow \beta_1 \mid \beta_2$

General types of parser



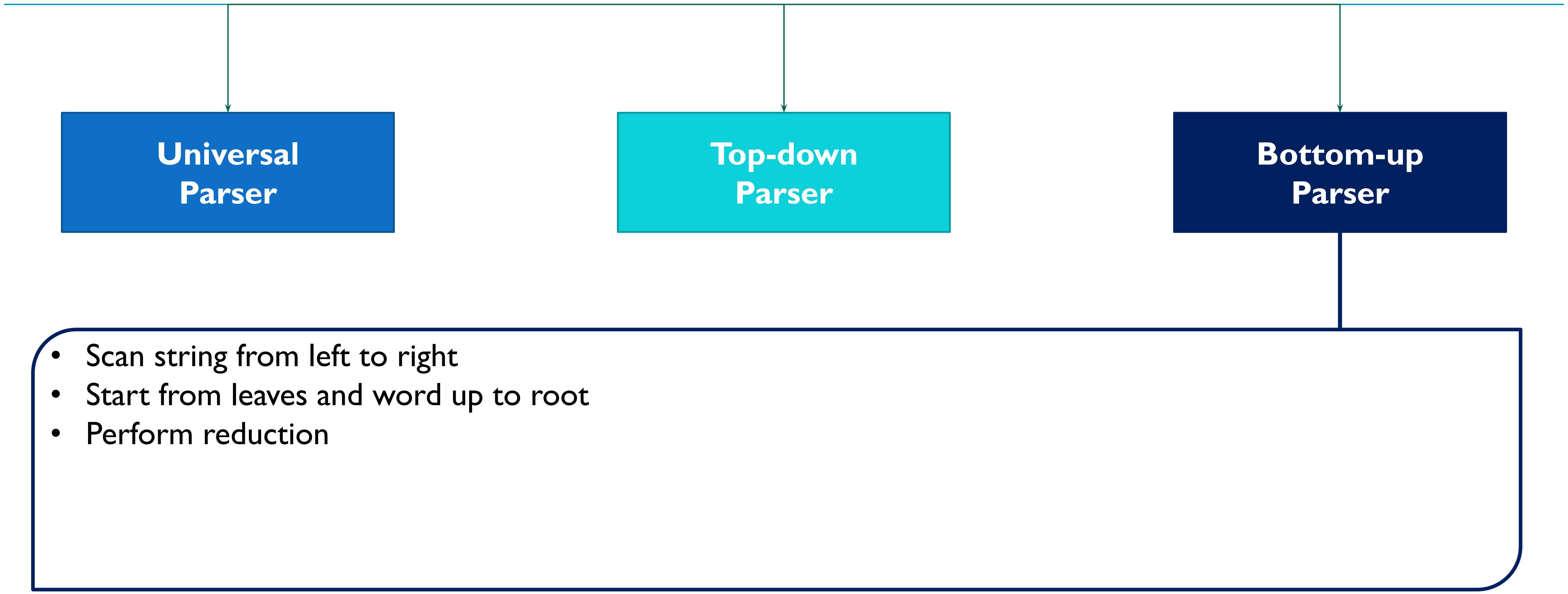
- General method
- Can parse any grammar
- Methods such as
  - Cocke-Younger-Kasami algorithm
  - Earley's algorithm

General types of parser



- Scan string from left to right
- Build parse tree from top (root) to the bottom (leaves)
- Perform derivation

General types of parser

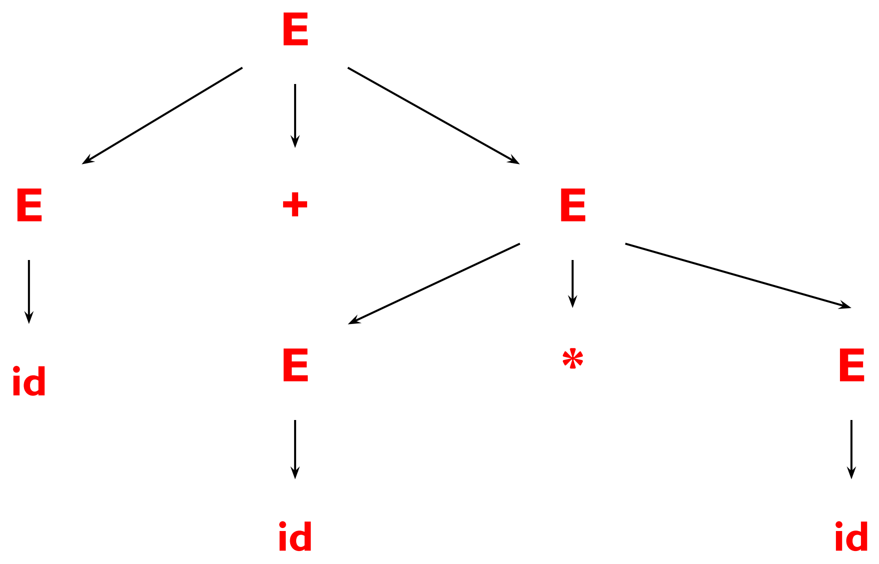


- Scan string from left to right
- Start from leaves and word up to root
- Perform reduction

Top-Down Parsing

- Construct parse tree for the input string starting from root and creating the nodes of parse tree in preorder (derivation)

- Grammar (G) : E → E + E | E * E | - E | (E) | id
- String : id + id * id



Different Top-Down Parsing Techniques

1. Recursive-Decent Parsing (RDP)
2. Predictive Parsing

1. Recursive-Decent Parsing (RDP)

- Require backtracking to find correct production to be applied
- Left recursive grammar can cause RDP to go into an infinite loop

1. Recursive-Decent Parsing (RDP)

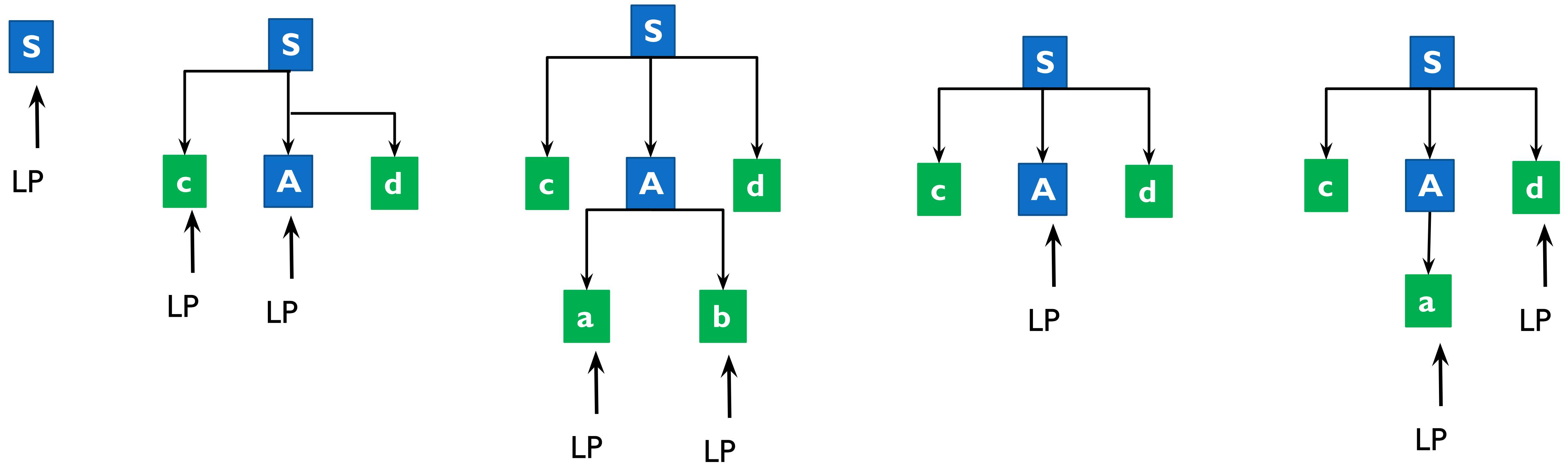
- Algorithm

```
void A( )
{
  choose an A-production, A→$X_1, X_2, \ldots, X_k$ ;
  for ( i = 1 to k)
  {
       if ( $X_i$ is a nonterminal)
            call procedure $X_i$( );
       else if ( $X_i$ equals the current input symbol $\alpha$ )
            advance the input to the next symbol;
       else
            /*  error occurred  */ ;
  }
  }
```

1. Recursive-Decent Parsing (RDP)

- Process:
  - ❖ Maintain 2 pointer
    - Lookahead pointer (LP) (point to top element of stack)
    - Input pointer (IP) (point to symbol in input string)
  - ❖ If nonterminal in stack (pointed by LP) then
    replace it by its production, and LP point to left most symbol in production
  - ❖ If terminal in stack (pointed by LP) then
    compare stack and input (pointed by LP and IP)
    - If match then
      advance both pointers (LP and IP)
    - If not match then
      backtrack

1. Recursive-Decent Parsing (RDP)



Grammar:

S → c A d
A → a b | a

String : **c** **a** **d**

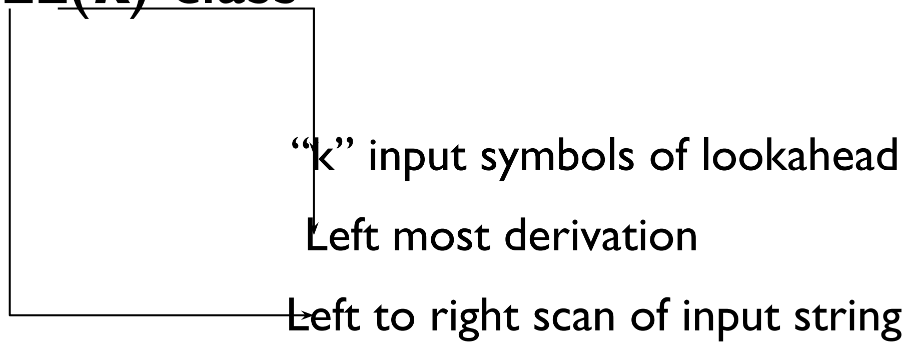
IP IP IP

String Match

backtrack

2. Predictive Parsing

- Specific case of RDP
- No backtracking is required
- Choose the correct production by looking ahead at the input a fixed number of symbols
- A class of grammar for which predictive parser can be constructed with looking ***k*** symbols ahead in the input is called LL(***k***) class


"k" input symbols of lookahead
Left most derivation
Left to right scan of input string

2. Predictive Parsing

- LL(1) grammar
  - ❖ Cover most programming constructs
  - ❖ Properties
    - Unambiguous
    - No left-recursion

FIRST and FOLLOW

- Used to construct top-down and bottom-up parser

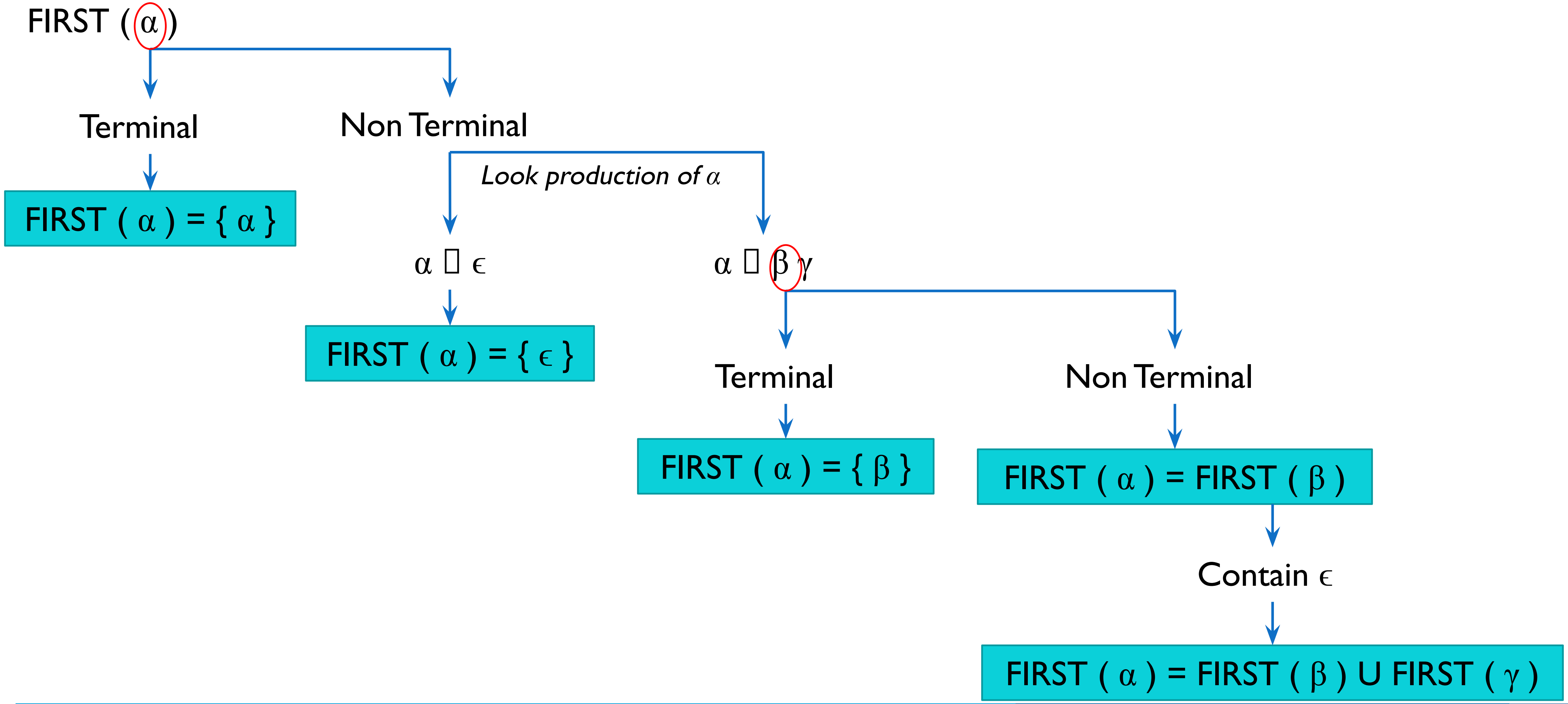
- FIRST (α) :

Set of terminals that begin strings derived from α

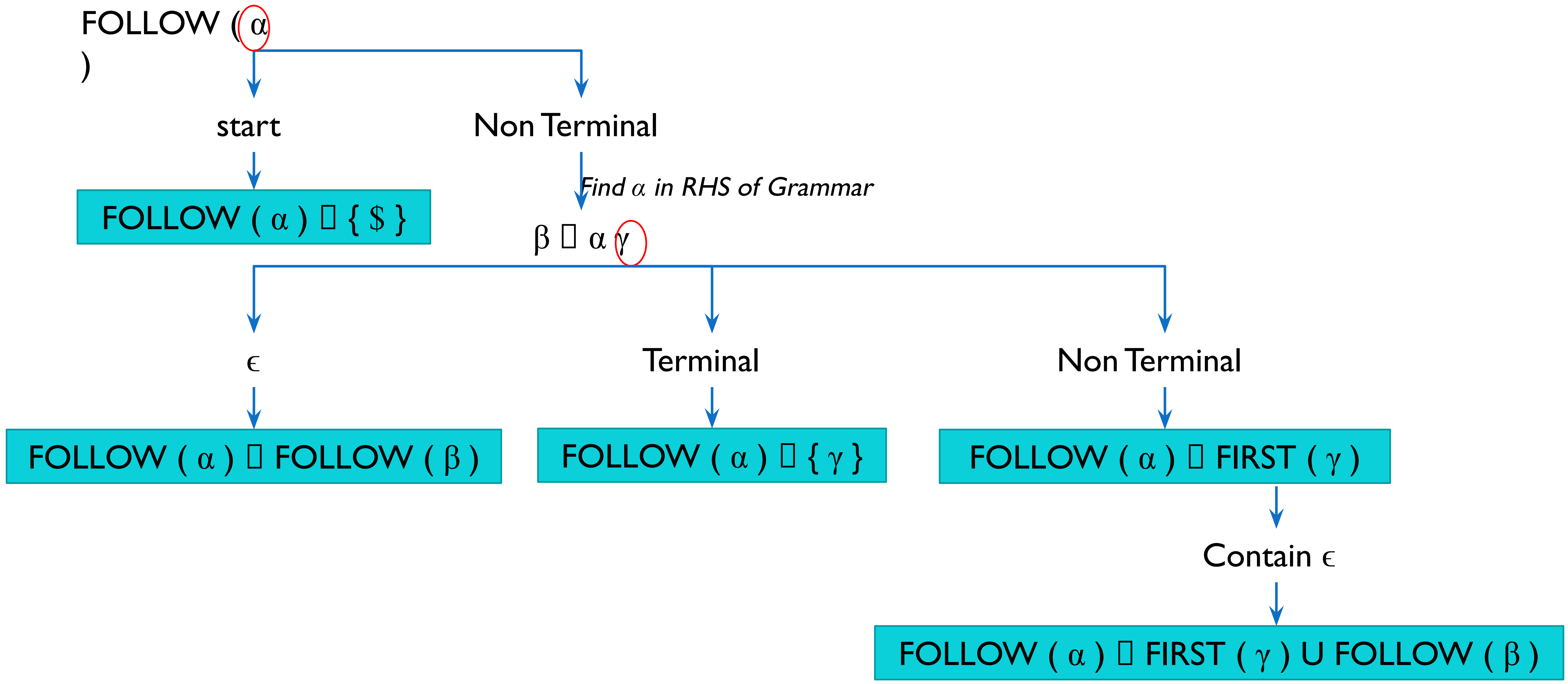
- FOLLOW (α) :

Set of terminals that can appear immediately to the right of α

FIRST


FIRST (α)
Terminal
FIRST (α) = { α }
Non Terminal
Look production of α
α → ϵ
FIRST (α) = { ϵ }
α → β γ
Terminal
FIRST (α) = { β }
Non Terminal
FIRST (α) = FIRST (β)
Contain ϵ
FIRST (α) = FIRST (β) U FIRST (γ)

FOLLOW



FOLLOW (α)
start
Non Terminal
FOLLOW (α) { $ }
Find α in RHS of Grammar
β α γ
ϵ
Terminal
Non Terminal
FOLLOW (α) FOLLOW (β)
FOLLOW (α) { γ }
FOLLOW (α) FIRST (γ)
Contain ϵ
FOLLOW (α) FIRST (γ) U FOLLOW (β)

FIRST and FOLLOW

Grammar:

E T E’
E’   + T E’ | ϵ
T F T’
T’ * F T’ | ϵ
F (E) | **id**

FIRST(E) = { id , (}
FIRST (E’) = { + , ϵ }
FIRST (T) = { id , (}
FIRST (T’) = { * , ϵ }
FIRST (F) = { id , (}

FOLLOW (E) = { $,) }
FOLLOW (E’) = { $,) }
FOLLOW (T) = { $,) , + }
FOLLOW (T’) = { $,) , + }
FOLLOW (F) = { $,) , + , * }

FIRST and FOLLOW

Grammar:
S → aBDh
B → cC
C → bC / ∈
D → EF
E → g / ∈
F → f / ∈

First(S) = { a }
First(B) = { c }
First(C) = { b , ∈ }
First(D) = { First(E) – ∈ } ∪ First(F) = { g , f , ∈ }
First(E) = { g , ∈ }
First(F) = { f , ∈ }

Follow(S) = { $ }
Follow(B) = { First(D) – ∈ } ∪ First(h) = { g , f , h }
Follow(C) = Follow(B) = { g , f , h }
Follow(D) = First(h) = { h }
Follow(E) = { First(F) – ∈ } ∪ Follow(D) = { f , h }
Follow(F) = Follow(D) = { h }

FIRST and FOLLOW

Grammar:
$S \rightarrow A$
$A \rightarrow aB \ / \ Ad$
$B \rightarrow b$
$C \rightarrow g$

First(S) = First(A) = { a }
First(A) = { a }
First(A’) = { d , ∈ }
First(B) = { b }
First(C) = { g }

Grammar after elimination of left recursion:
$S \rightarrow A$
$A \rightarrow aBA'$
$A' \rightarrow dA' \ / \ \in$
$B \rightarrow b$
$C \rightarrow g$

Follow(S) = { $ }
Follow(A) = Follow(S) = { $ }
Follow(A’) = Follow(A) = { $ }
Follow(B) = { First(A’) – ∈ } ∪ Follow(A) = { d , $ }
Follow(C) = NA

2. Predictive Parsing

- Remove Left recursion from the grammar
- Convert it to Left factored grammar
- How to construct Predictive Parsing Table

❖ Find FIRST and FOLLOW set for all nonterminals after removal of left recursion and conversion to left factored grammar

❖ If "a" is in FIRST(X) then write production of X which can derive Xaβ in [X , a]

❖ If "ϵ" is in FIRST(X) then see the follow set of X
place production (which give ϵ in FIRST(X)) in all α ∊ FOLLOW(X)

2. Predictive Parsing

Grammar:

E T E'

E' + T E' | ϵ

T F T'

T' * F T' | ϵ

F (E) | **id**

FIRST(E) = { id , (}
FIRST (E') = { + , ϵ }
FIRST (T) = { id , (}
FIRST (T') = { * , ϵ }
FIRST (F) = { id , (}

FOLLOW (E) = { $,) }
FOLLOW (E') = { $,) }
FOLLOW (T) = { $,) , + }
FOLLOW (T') = { $,) , + }
FOLLOW (F) = { $,) , + , * }

Nonterminal	Terminal					
	id	+	*	(	)	$
E	TE'			TE'		
E'		+TE'			ϵ	ϵ
T	FT'			FT'		
T'		ϵ	*FT'		ϵ	ϵ
F	id			(E)		

All cell contain one and only one production so grammar is LL(1)

2. Predictive Parsing

(1) Parse the string id+id

STACK	INPUT	OUTPUT
$ E	id + id $	
$ E' T	id + id $	E → T E'
$ E' T' F	id + id $	T → F T'
$ E' T' id	id + id $	F → id
$ E' T'	+ id $	
$ E'	+ id $	T' → ϵ
$ E' E' T +	+ id $	E' → + T E'
$ E' E' T	id $	
$ E' E' T' F	id $	T → F T'
$ E' E' T' id	id $	F → id
$ E' E' T'	$	
$ E' E'	$	T' → ϵ
$ E'	$	E' → ϵ
$	$	E' → ϵ

2. Predictive Parsing

(2) Parse the string (id+id)*id

STACK	INPUT	OUTPUT
$ E	(id + id) * id $	
$ E' T	(id + id) * id $	E→TE'
$ E' T' F	(id + id) * id $	T→FT'
$ E' T') E (	(id + id) * id $	F→(E)
$ E' T') E	id + id) * id $	
$ E' T') E' T	id + id) * id $	(E)→TE'
$ E' T') E' T' F	id + id) * id $	T→FT'
$ E' T') E' T' id	id + id) * id $	F→id
$ E' T') E' T'	+ id) * id $	
$ E' T') E'	+ id) * id $	T'→ϵ
$ E' T') E' T +	+ id) * id $	E'→+TE'
$ E' T') E' T	id) * id$	
$ E' T') E' T' F	id) * id$	T→FT'
$ E' T') E' T' id	id) * id$	F→id
$ E' T') E' T'	) * id$	
$ E' T') E'	) * id$	T'→ϵ
$ E' T')	) * id$	E'→ϵ
$ E' T'	* id$	
$ E' T' F*	* id$	T'→*FT'
$ E' T' F	id$	
$ E' T' id	id$	F→id
$ E' T'	$	
$ E'	$	T'→ϵ
$	$	E'→ϵ

2. Predictive Parsing

Grammar:

be *be* **or** *bt* | *bt*

bt *bt* **and** *bf* | *bf*

bf **not** *bf* | (*be*) | **true** | **false**

FIRST(*be*) = { **not** , (, **true** , **false** }
FIRST (B') = { **or** , ϵ }
FIRST (*bt*) = { **not** , (, **true** , **false** }
FIRST (A') = { **and** , ϵ }
FIRST (*bf*) = {**not** , (, **true** , **false** }

FOLLOW (*be*) = { $,) }
FOLLOW (B') = { $,) }
FOLLOW (*bt*) = { $,) , **or** }
FOLLOW (A') = { $,) , **or** }
FOLLOW (*bf*) = { $,) , **or** , **and** }

Remove left recursion

Grammar:

be *bt* B'

B' **or** *bt* B' | ϵ

bt *bf* A'

A' **and** bf A' | ϵ

bf **not** *bf* | (*be*) | **true** | **false**

Nonterminal	Terminal							
	or	**and**	**not**	(	)	**true**	**false**	**$**
be			bt B'	bt B'		bt B'	bt B'	
B'	or bt B'				ϵ			ϵ
bt			bf A'	bf A'		bf A'	bf A'	
A'	ϵ	and bf A'			ϵ			ϵ
bf			not bf	(be)		true	false	

All cell contain one and only one production so grammar is LL(1)

2. Predictive Parsing

Grammar:

S → i E t S S' | a

S' → e S | ϵ

E → b

FIRST (S) = { i , a }
FIRST (S') = { e , ϵ }
FIRST (E) = { b }

FOLLOW (S) = { e , $ }
FOLLOW (S') = { e , $ }
FOLLOW (E) = { t }

Nonterminal	Terminal					
	i	t	a	e	b	$
S	i E t S S'		a			
S'				e S ϵ		ϵ
E					b	

Multiple production in cell so grammar is **not** LL(1)

2. Predictive Parsing

Grammar:

S (L) | a
L L , S | S

FIRST (S) = { (, a }
FIRST (L) = { (, a }
FIRST (L') = { , , ϵ }

FOLLOW (S) = { $, , ,) }
FOLLOW (L) = {) }
FOLLOW (L') = {) }

Remove left recursion

Grammar:

S (L) | a
L S L'
L' , S L' | ϵ

Nonterminal	Terminal				
	(	)	a	,	$
S	(L)		a		
L	S L'		S L'		
L'		ϵ		, S L'	

All cell contain one and only one production so grammar is LL(1)

2. Predictive Parsing

Grammar:

D □ *type* *list* ;
list □ *list* , **id** | **id**
type □ **int** | **float**

Space

FIRST (D) = { int , float}
FIRST (list) = { id }
FIRST (L') = { , , ϵ}
FIRST (type) = { int , float }

FOLLOW (D) = { $ }
FOLLOW (list) = { ; }
FOLLOW (L') = { ; }
FOLLOW (type) = { ' ' }

Space

Remove left recursion

Grammar:

D □ *type* *list* ;
list □ **id** L'
L' □ , id L' | ϵ
type □ **int** | **float**

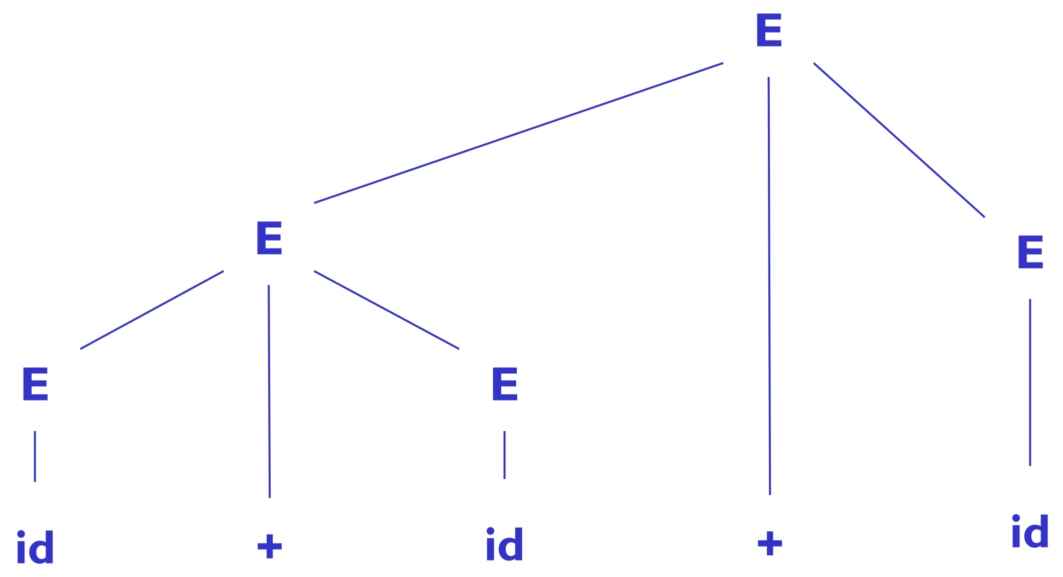
Nontermina l	Terminal						
	;	**id**	,	**int**	**float**	' '	**$**
D				type list ;	type list ;		
list		id L'					
L'	ϵ		, id L'				
type				int	float		

All cell contain one and only one production so grammar is LL(1)

Bottom-Up Parsing

- Construct parse tree for the input string starting at the leaves (bottom) and working up towards the root (top) (reduction)

- Grammar (G) : E E + E | E * E | - E | (E) | id
- String : id + id + id



Bottom-Up Parsing

- Handle
  - ❖ Handle of the string is a substring that matches with RHS of production whose reduction by LHS of production represents one step along the reverse of a right most derivation
  - ❖ If then production in the portion following α is a handle of αβw

Grammar : E → E + T | T
T → T * F | F
F → (E) | id

String: id * id

String	Handle	Reducing production
id * id	id	F → id
F * id	F	T → F
T * id	id	F → id
T * F	T * F	T → T * F
T	T	E → T

Bottom-Up Parsing

- Handle Pruning
  - A right most derivation in reverse can be obtain by handle pruning

Different Bottom-Up Parsing Techniques

1. Shift-Reduce Parsing
2. Operator Precedence Parsing
3. LR Parsing
   1) Simple LR (SLR or LR(0))
   2) Canonical LR (CLR or LR(1))
   3) Lookahead LR (LALR)

1. Shift-Reduce Parsing

- Stack holds grammar symbols
- Input buffer holds the string to be parsed
- Handle always appears at the top of the stack
- Use $ to mark bottom of the stack and also the right end of the input
- Process:
  - ❖ During left to right scan of input string, shift zero or more input symbols onto the stack, until it is ready to reduce a string β
  - ❖ The reduce β to the head (LHS) of the appropriate production
  - ❖ Repeats this cycle until detect error or until stack contain start symbol and input is empty

1. Shift-Reduce Parsing

- There are 4 possible actions
  - ❖ Shift
    - Shift the next input symbol onto the top of the stack
  - ❖ Reduce
    - Replace handle with LHS in the stack
  - ❖ Accept
    - Parsing complete successfully
  - ❖ Error
    - Discover a syntax error and call an error recovery routine

1. Shift-Reduce Parsing

Grammar :

E → E + E | E * E | id

String:

id + id * id

Stack	Input	Action

1. Shift-Reduce Parsing

Grammar :

E → E + E | E * E | id

String:

id + id * id

Stack	Input	Action
$	id + id * id $	Shift
$ id	+ id * id $	Reduce E → id
$ E	+ id * id $	Shift
$ E +	id * id $	Shift
$ E + id	* id $	Reduce E → id
$ E + E	* id $	Shift
$ E + E *	id $	Shift
$ E + E * id	$	Reduce E → id
$ E + E * E	$	Reduce E→ E * E
$ E + E	$	Reduce E → E + E
$ E	$	Accept

1. Shift-Reduce Parsing

- Conflict during shift reduce parsing

  - ❖ Shift / reduce conflict
    - Cannot decide whether to shift or to reduce

  - ❖ Reduce / reduce conflict
    - Cannot decide which of several reduction to make

2. Operator Precedence Parsing

- Operator grammar
  - ❖ The grammar has the property (among other essential requirements) that no production right side is ϵ or has two adjacent nonterminals.

E.g. E □ EAE | (E) | -E | **id**
A □ + | - | * | / | ^

Not a operator grammar as EAE as consecutive nonterminals

E □ E+E | E-E | E*E | E/E | E^E | (E) | -E | **id**

Equivalent operator grammar

2. Operator Precedence Parsing

- Define precedence relation between pair of terminals by disjoint relation symbols ⋗,⋖ and ≐
- If α1 and α2 are operators or terminal symbols,

❖ If operator α1 has higher precedence than α2 then α1 ⋗ α2 and α2 ⋖ α1

❖ If α1 and α2 are operators of equal precedence than α1 ⋗ α2 and α2 ⋗ α1 if operators are left associative, or make α1 ⋖ α2 and α2 ⋖ α1 if operators are right associative.

- α1 ⋗ α2 and α2 ⋗ α1 are not same always.

α ⋖ id	id ⋗ α	α ⋖ (	(⋖ α
) ⋗ α	α ⋗)	α ⋗ $	$ ⋖ α

***For all operators α

(≐)	$ ⋖ (	$ ⋖ id
(⋖ (	id ⋗ $	) ⋗ $
(⋖ id	Id ⋗)	) ⋗

2. Operator Precedence Parsing

- How to parse string (using operator precedence table)

1. Construct operator precedence table
2. Place $ (imaginary terminal marking) at staring and ending of string (mark each end of string)
3. Put relation between each symbols in string
4. Scan the string form left to right until first $\gtrdot$ is encounter
5. Then scan back over any $\doteq$ until $\lessdot$ is encounter
6. Handle is every thing between $\lessdot$ and $\gtrdot$ reduce to LHS of appropriate production

2. Operator Precedence Parsing

Grammar:

E → E + E | E * E | id

operator precedence table

Right side

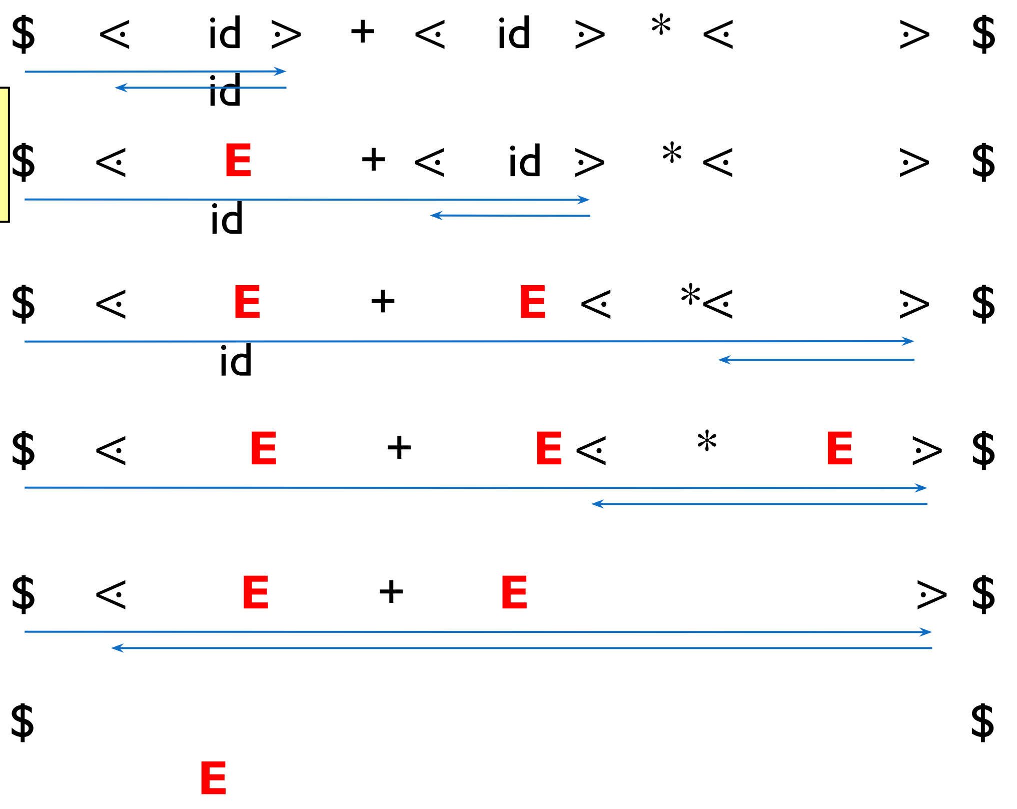
Lef t sid e

	id	+	*	$
id		⋗	⋗	⋗
+	⋖	⋗	⋖	⋗
*****	⋖	⋗	⋗	⋗
$	⋖	⋖	⋖	

Left + has high priority then right +

id is replaced with E
Now compare $ + id * id $

String:



2. Operator Precedence Parsing

- Algorithm : operator precedence parsing

❖ Method :

Initially the stack contains \$ and the input buffer the string *w*\$. To parse we execute the below program

```
1.  Set ip to point to the first symbol of w$:
2.  Repeat forever
3.      if $ is on top of the stack and ip points to $ then
4.          return
5.      else begin
6.          let a be the topmost terminal symbol on the stack and b be the symbol pointed by ip
7.          if a ⋖ b or a ≐ b the begin
8.              push b onto the stack
```

2. Operator Precedence Parsing

- Algorithm : operator precedence parsing

❖ Method :

```
9.          advance ip to the next input symbol
10.      end
11.      else  if a ⋗ b then
12.          repeat
13.              pop the stack
14.          until the top stack terminal is related by ⋖ to the terminal most recently popped
15.      else
16.          error()
•    end
```

2. Operator Precedence Parsing

- Operator precedence function
  - ❖ Precedence between "a" and "b" can be determined by numerical comparison function f and g
  - ❖ $f(a) = g(b)$ if $a \doteq b$
  - ❖ $f(a) < g(b)$ if $a \lessdot b$
  - ❖ $f(a) > g(b)$ if $a \gtrdot b$

2. Operator Precedence Parsing

- Algorithm : construct precedence functions

❖ Input :

An operator precedence matrix

❖ Output :

Precedence functions representing the input matrix, or an indication that none exist

2. Operator Precedence Parsing

- Algorithm : construct precedence functions

- **Method** :

1. Create symbol "f_a" and g_a" for each terminal "a" and \$
2. Partitions the created symbols into as many group as possible in a such a way that if $a \doteq b$ then "f_a" & "g_b" are in same group
3. Create a directed graph whose nodes are the groups found in step-2
   for any "a" and "b"
   if $a \lessdot b$ then place an edge from group "g_b" to group "f_a"
   if $a \gtrdot b$ then place an edge from group "f_a" to group "g_b"
4. If graph is constructed in step-3 has a cycle then no precedence function exist. If there are no cycle then let f(a) be the length of the longest path beginning at the group of "f_a" and g(a) be the length of the longest path from beginning at the group of "g_a"

2. Operator Precedence Parsing

Right side

------- g -------

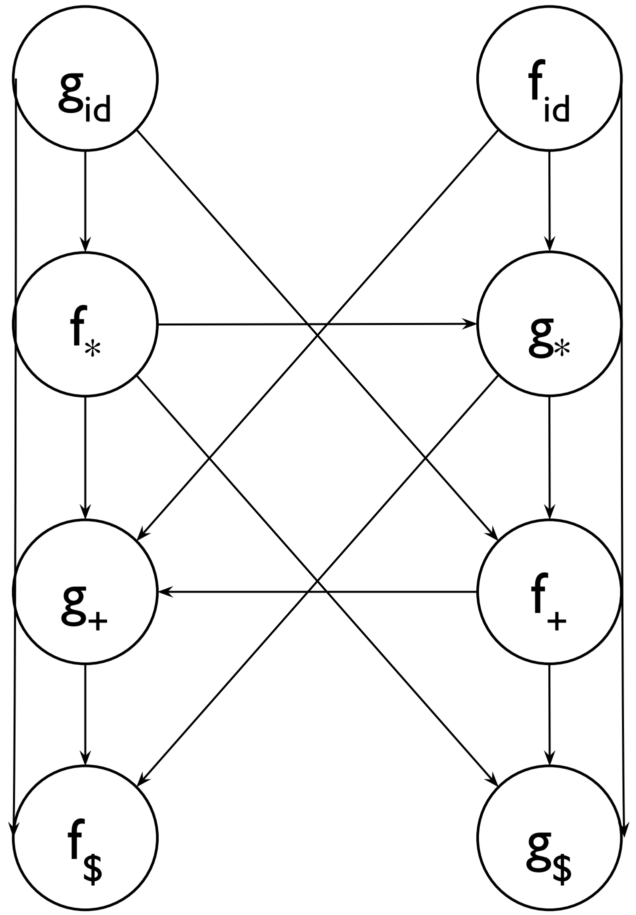
Lef
t
side

-- f

--

	id	+	*	$
id		$\gtrdot$	$\gtrdot$	$\gtrdot$
+	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
*****	$\lessdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
$	$\lessdot$	$\lessdot$	$\lessdot$	

Draw edge from grater to less
e.g. F(+) > g(+) so edge from f+ to g+



Find max path to reach either f$ or g$

	id	+	*	$
f	4	2	4	0
g	5	1	3	0

2. Operator Precedence Parsing

Parse string id + id * id

$	id	+	id	*	id	$
0	5	2	5	4	5	0
$	E	+	E	*	E	$
0		2		4		0
$	E	+	E			$
0		2				0
$	E					$
0						0

	id	+	*	$
f	4	2	4	0
g	5	1	3	0

2. Operator Precedence Relations from Associativity and Precedence

Rule 1: If operator

2. Operator Precedence Parsing

Grammar:

E E + E | E – E | E * E | E / E | E ^ E | (E) | id

operator precedence table

Right side

Lef t sid e

	id	**+**	**-**	*****	**/**	**^**	**(**	**)**	**$**
id		⋗	⋗	⋗	⋗	⋗		⋗	⋗
+	⋖	⋗	⋗	⋖	⋖	⋖	⋖	⋗	⋗
-	⋖	⋗	⋗	⋖	⋖	⋖	⋖	⋗	⋗
*****	⋖	⋗	⋗	⋗	⋗	⋖	⋖	⋗	⋗
/	⋖	⋗	⋗	⋗	⋗	⋖	⋖	⋗	⋗
^	⋖	⋗	⋗	⋗	⋗	⋖	⋖	⋗	⋗
(	⋖	⋖	⋖	⋖	⋖	⋖	⋖	≐	
)		⋗	⋗	⋗	⋗	⋗		⋗	⋗
$	⋖	⋖	⋖	⋖	⋖	⋖	⋖		

3. LR parsing

- LR : left to right scan Right most derivation
- Can handle left recursive grammar
- Can not handle the ambiguous grammar

3. LR parsing

- 3 types
  - Simple LR (SLR)
    - Can solve LR(0) grammars
  - Canonical LR (CLR)
    - Can solve LR(1) grammars
  - Lookahead LR (LALR)

3. LR parsing : (1) SLR

- Weakest methods from all 3 LR methods

- Process
  - ❖ Convert grammar in augmented grammar by adding one more starting symbol S' → S
  - ❖ Construct LR(0) item sets
  - ❖ Construct parsing table

3. LR parsing : (1) SLR

- How to construct item set

❖ Production A → XYZ can have 4 forms

A → . XY Z

A → X .Y Z

A → XY . Z

A → XY Z .

❖ Production A →ϵ can generate only one item A → .

3. LR parsing : (1) SLR

 How to construct item set

❖ Closure function

 If **I** is a set of items for a grammar **G** then closure(**I**) is constructed by two rules

1. Initially every item in **I** is in closure of **I**
2. A α . B β is in closure(**I**) & B γ is the production rule then B . γ is added in closure(**I**) continue this rule till no new item can be added

❖ GOTO function

 goto(**I** , x) where "**I**" is item set and "x" is grammar symbol

 goto(**I** , x) = closure of the set of all items [A α x . β] such that [A α . x β] is in **I**

3. LR parsing : (1) SLR

- How to construct parsing table

1. If S’→S**.** is in **I**i then set action[i , $] to accept

2. If A → α **.** x β is in **I**i , where “x” is terminal , and goto(**I**i , x)= **I**j then set action[i , x] to “shift j”

3. If A → α **.** B β is in **I**i, where B is nonterminal then set goto[i , B] to the j where we are having first production A → α B **.** β in **I**j

4. If A → α **.** is in **I**i then set action[i , x] to “reduce A→α” for all “x” in follow(A)

3. LR parsing : (1) SLR

Grammar:

E → E + T | T
T → T * F | F
F → (E) | **id**

Make augmented grammar

Augmented Grammar:

E ' → E
E → E + T | T
T → T * F | F
F → (E) | **id**

Added new start symbol E '

Give numbers to production (will need it in table generation)

0) E ' → E
1) E → E + T
2) E → T
3) T → T * F
4) T → F
5) F → (E)
6) F → id

Find follow set for all nonterminal

FOLLOW(E) = { $,) , +}
FOLLOW(T) = { $,) , + , * }
FOLLOW(F) = { $,) , + , * }

Its same is I2 so don't give new name, give name I2
Same ous item set
give

Production with ". E" become "E ."

3. LR parsing : (1) SLR

Construct LR(0) item sets

Io = E ’ . E
E . E + T
E . T
T . T * F
T . F
F . (E)
F . **id**

I1 = goto(I0 , E) = E ’ E .
E E . + T

I2 = goto(I0 , T) = E T .
T T . * F

I3 = goto(I0 , F) = T F .

I4 = goto(I0 , ( ) = F (. E)
E . E + T
E . T
T . T * F
T . F
F . (E)
F . **id**

I5 = goto(I0 , **id**) = F **id** .

I6 = goto(I1 , +) = E E + . T
T . T * F
T . F
F . (E)
F . **id**

I7 = goto(I2 , *) = T T * . F
F . (E)
F . **id**

I8 = goto(I4 , E) = F (E .)
E E . + T

I2 = goto(I4 , T) = E T .
T T . * F

I3 = goto(I4 , F) = T F .

I4 = goto(I4 , ( ) = F (. E)
E . E + T
E . T
T . T * F
T . F
F . (E)
F . **id**

I5 = goto(I4 , id) = F **id** .

New item set so give new name

3. LR parsing : (1) SLR

Construct LR(0) item sets

I9 = goto(I6 ,T) = E → E + T .
T → T . * F

I3 = goto(I6 , F) = T → F .

I4 = goto(I6 , ( ) = F → (. E)
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → . id

I5 = goto(I6 , id) = F → id .

I10 = goto(I7 , F) = T → T * F .

I4 = goto(I7 , ( ) = F → (. E)
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → . id

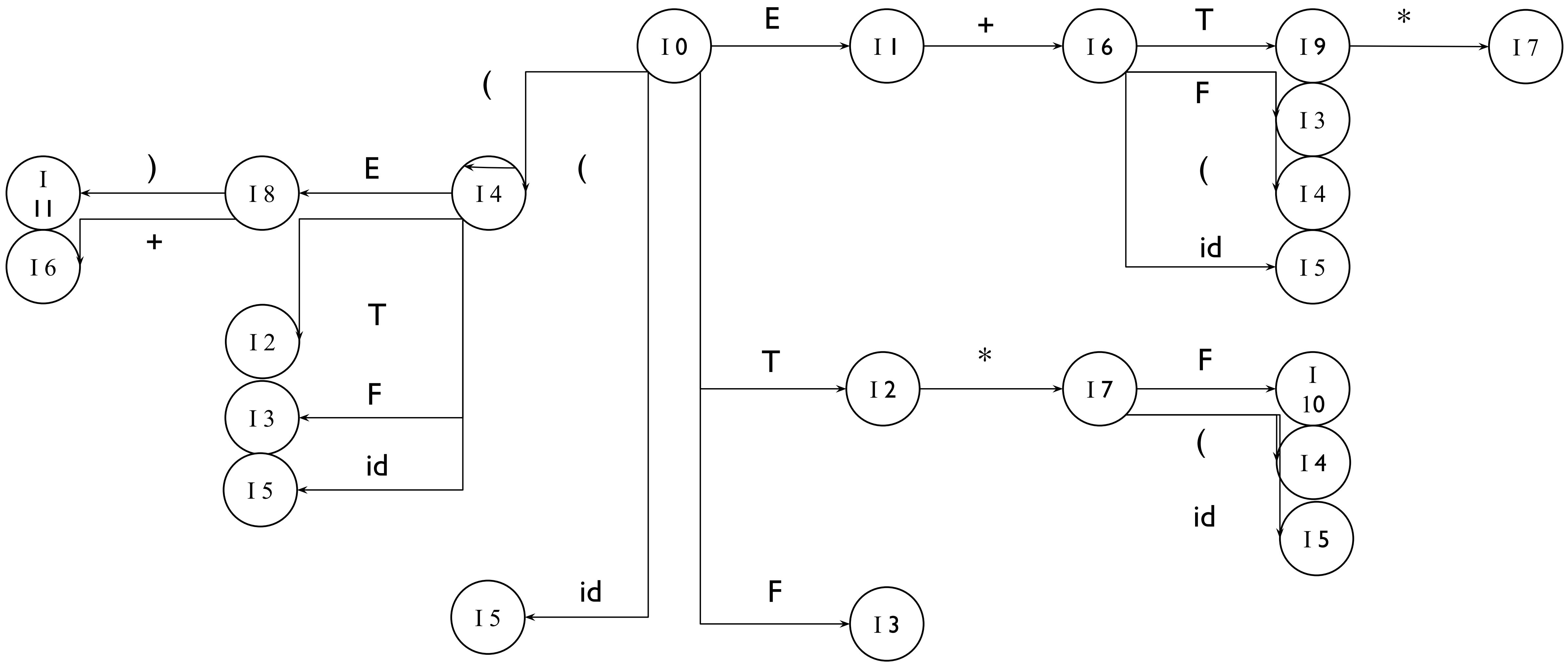
I5 = goto(I7 , id) = F → id .

I11 = goto(I8 ,)) = F → (E) .

I6 = goto(I8 , +) = E → E + . T
T → . T * F
T → . F
F → . (E)
F → . id

I7 = goto(I9 , *) = T → T * . F
F → . (E)
F → . id

3. LR parsing : (1) SLR



I3 has item T□F. (dot at the end)
Production number of T□F is 4
Follow(T)={+,*,),$}
In (3,+) (3,*) (3,)) (3,$) do entry of R4 (reduce 4)

3. LR parsing : (1) SLR

Item set	action						goto		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				Acc			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

3. LR parsing : (1) SLR

Stack	Input	Action
0	id * id + id $	Shift
0 id 5	* id + id $	Reduce by F □ id
0 F 3	* id + id $	Reduce by T □ F
0 T 2	* id + id $	Shift
0 T 2 * 7	id + id $	shift
0 T 2 * 7 id 5	+ id $	Reduce by F □ id
0 T 2 * 7 F 10	+ id $	Reduce by T □ T * F
0 T 2	+ id $	Reduce by E □ T
0 E 1	+ id $	Shift
0 E 1 + 6	id $	Shift
0 E 1 + 6 id 5	$	Reduce by F □ id
0 E 1 + 6 F 5	$	Reduce by T □ F
0 E 1 + 6 T 9	$	Reduce by E □ E + T
0 E 1	$	Accept

In parsing table entry of (0,id) is S5
So Action is Shift
Shift one element from input to stack
Place 5 after that in stack

In parsing table entry of (3,*) is R6
So Action is Reduce with production 6 (F□id)
In stack find id and replace with F
In stack it become 0F
In parsing table entry of (0,F) is 3
Place 3 in stack

3. LR parsing : (2) CLR

- Process
  - ❖ Convert grammar in augmented grammar by adding one more starting symbol S’ → S
  - ❖ Construct LR(1) item sets
  - ❖ Construct parsing table

3. LR parsing : (2) CLR

- How to construct item set

❖ Closure function

- If **I** is a set of items for a grammar **G** then closure(**I**) is constructed by two rules

1. Initially every item in **I** is in closure of **I**
2. $A \rightarrow \alpha\ .\ B\ \beta\ ,\ a$ is in closure(**I**) & $B \rightarrow \gamma$ is the production rule
   then $B \rightarrow .\ \gamma\ ,\ \mathrm{FIRST}(\beta a)$ is added in closure(**I**) continue this rule till no new item can be added

❖ GOTO function

- goto(**I** , x) where "**I**" is item set and "x" is grammar symbol
- goto(**I** , x) = closure of the set of all items $[\,A \rightarrow \alpha\ x\ .\ \beta\ ,\ a\,]$ such that $[A \rightarrow \alpha\ .\ x\ \beta\ ,\ a\,]$ is in **I**

3. LR parsing : (2) CLR

- How to construct parsing table

1. If S’→S. , $ is in **I**i then set action[i , $] to accept

2. If A → α . x β , a is in **I**i , where “x” is terminal , and goto(**I**i , x)= **I**j then set action[i , x] to “shift j”

3. If A → α . B β is in **I**i, where B is nonterminal then set goto[i , B] to the j where we are having first production A → α B . β , a in **I**j

4. If A → α . , a is in **I**i then set action[i , a] to “reduce A→α”

3. LR parsing : (2) CLR

Grammar:

S → CC
C → cC | d

Make augmented grammar

Augmented Grammar:

S ' → S
S → CC
C → cC | d

Give numbers to production (will need it in table generation)

0) S' → S
1) S → CC
2) C → cC
3) C → d

Find first set for all nonterminal

FIRST (S) = { c , d }
FIRST (C) = { c , d }

S’→.S need to compare with A → α.Bβ , a
here β is ϵ and a is $, so FIRST(βa) = {$}
$ is added in look ahead of S → .CC

3. LR parsing : (2) CLR

Construct LR(1) item sets

Io = S ’ → . S , $
S → . CC , $
C → .cC , c/d
C → .d , c/d

I1 = goto(I0 , S) = S ’ → S. , $

I2 = goto(I0 , C) = S → C.C , $
C → .cC , $
C → .d , $

I3 = goto(I0 , c) = C → c.C , c/d
C → .cC , c/d
C → .d , c/d

I4 = goto(I0 , d) = C → d. , c/d

I5 = goto(I2 , C) = S → CC. , $

I6 = goto(I2 , c) = C → c.C , $
C → .cC , $
C → .d , $

I7 = goto(I2 , d) = C → d. , $

I8 = goto(I3 , C) = C → cC. , c/d

I3 = goto(I3 , c) = C → c.C , c/d
C → .cC , c/d
C → .d , c/d

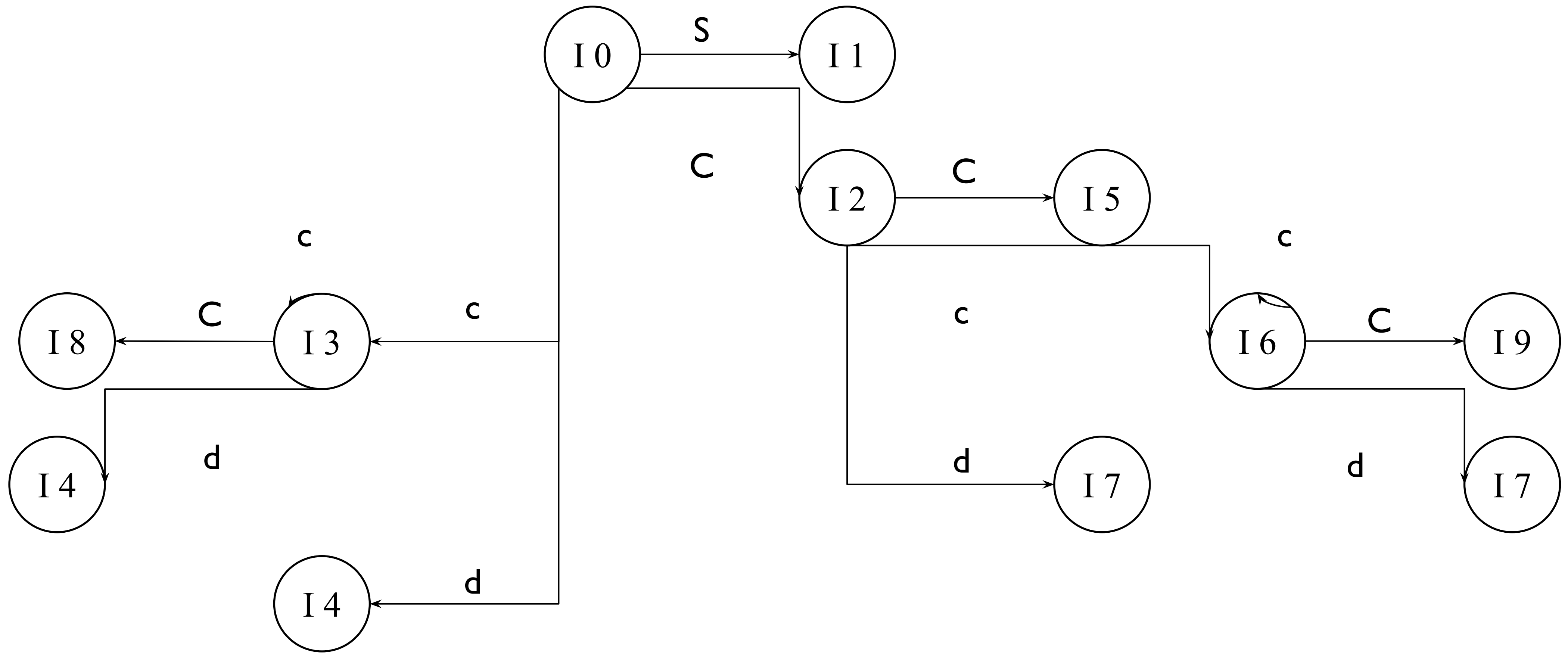
I4 = goto(I3 , d) = C → d. , c/d

I9 = goto(I6 , C) = C → cC. , $

I6 = goto(I6 , c) = C → c.C , $
C → .cC , $
C → .d , $

I7 = goto(I6 , d) = C → d. , $

3. LR parsing : (2) CLR



3. LR parsing : (2) CLR

I3 = goto (I0,c)
So do entry of shift3(S3) in (0,c)

I2 = goto (I0,C)
So do entry of 2 in (0,C)

I4 contain production C􀀀d.,c/d
Dot at the end so need to do reduce entry
C􀀀d has production number 3
Look ahead are c and d
So do entry of R3 in (4,c0 and (4,d)

Item set	action			goto	
	c	d	$	S	C
0	$\boxed{\text{S3}}$	S4		1	$\boxed{2}$
1			Acc		
2	S6	S7			5
3	S3	S4			8
4	$\boxed{\text{R3}}$	$\boxed{\text{R3}}$			
5			R1		
6	S6	S7			9
7			R3		
8	R2	R2			
9			R2		

3. LR parsing : (3) LALR

- Process
  - ❖ Convert grammar in augmented grammar by adding one more starting symbol S’ S
  - ❖ Construct LR(1) item sets
  - ❖ Combine the item sets having same core but different lookahead
  - ❖ Construct parsing table

3. LR parsing : (3) LALR

Grammar:

S CC
C cC | d

Make augmented grammar

Augmented Grammar:

S ’ S
S CC
C cC | d

Give numbers to production
(will need it in table generation)

0) S’ S
1) S CC
2) C cC
3) C d

Find first set for all nonterminal

FIRST (S) = { c , d }
FIRST (C) = { c , d }

3. LR parsing : (3) LALR

Construct LR(1) item sets

Io = S ’ . S , $
S . CC , $
C .cC , c/d
C .d , c/d

I1 = S ’ S. , $

I2 = S C.C , $
C .cC , $
C .d , $

I3 = C c.C , c/d
C .cC , c/d
C .d , c/d

I4 = C d. , c/d

I5 = S CC. , $

I36 = C c.C , c/d/$
C .cC , c/d/$
C .d , c/d/$

I47 = C d. , c/d/$

I6 = C c.C , $
C .cC , $
C .d , $

I7 = C d. , $

I8 = C cC. , c/d

I89 = C cC. , c/d/$

I9 = C cC. , $

3. LR parsing : (3) LALR

Item set	action			goto	
	c	d	$	S	C
0	S36	S47		1	2
1			Acc		
2	S36	S47			5
36	S36	S47			89
47	R3	R3	R3		
5			R1		
89	R2	R2	R2		

Using Ambiguous Grammar

- Every ambiguous grammar fails to be LR

- Certain types of ambiguous grammars are useful in the specification and implementation of languages

- Ambiguous grammar provides a shorter, more natural specification than any equivalent unambiguous grammar

Using Ambiguous Grammar

- □ Ambiguous grammar

  E □ E + E | E * E | (E) | id

- □ Equivalent unambiguous grammar

  E □ E + T | T

  T □ T * F | F

  F □ (E) | id

❖ Grammar is ambiguous because it does not specify the associativity and precedence of the operators + and *

❖ Generates same language but give + a lower precedence than * and makes both operators left associative

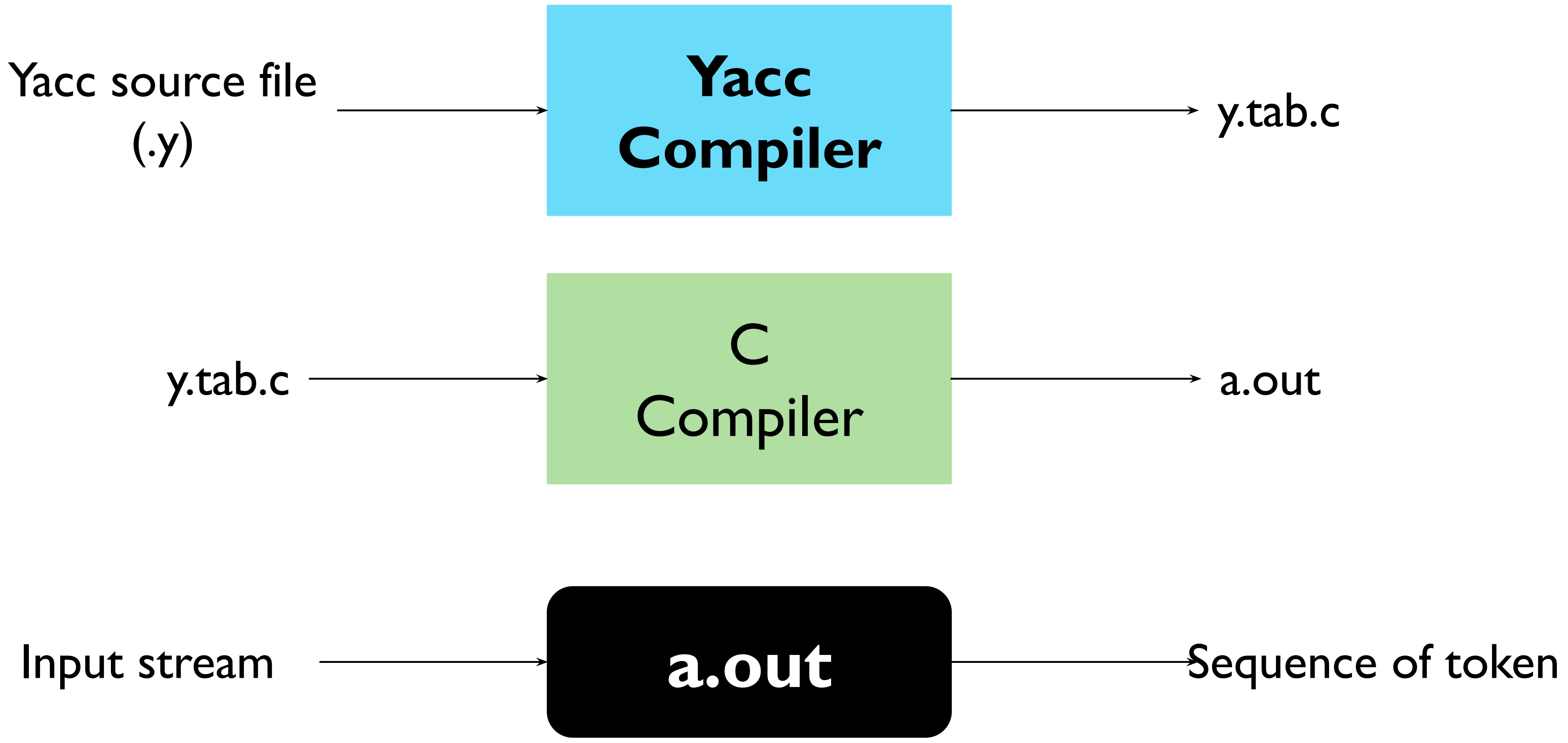
Using Ambiguous Grammar

- Reasons : might want to use ambiguous grammar instead of unambiguous

1. Can easily change the associativities and precedence levels of the operators without disturbing the production of ambiguous grammar or the number of states in the resulting parser.

2. Unambiguous grammar will spend a large fraction of time reducing by the productions $E \rightarrow T$ and $T \rightarrow F$.
   The parser for ambiguous grammar will not waste time reducing by these single productions.

Syntax Analyzer Generator YACC



Structure of Yacc Program

```
declarations
%%
translation rules
%%
Supporting c-routines
```